AN1776
APPLICATION NOTE

INTERRUPT HANDLING FOR STR7 MICROCONTROLLERS

by MCD Application Team

INTRODUCTION

An exception occurs when the normal flow of a program has to be halted temporarily. The reasons for an exception to occur can be due to an interrupt from a hardware peripheral, a coding error or incompatibility or even a user-defined exception. In each of these cases, the exception needs to be 'handled'. This means, the normal program execution needs to be paused, and the processor needs to be directed to a specially written section of code which performs a series of predetermined actions. The process of performing these special actions is called exception handling. This document gives a description of the use of the STR7 interrupt controller (EIC), as well as how to configure the EIC to be able to handle IRQ and FIQ exception, and therefore how it can be customized for your own application requirements.

All examples provided with this application note are developed with RVDK 2.1 for ST and optimized for the STR710-Demoboard.

This document assumes that the reader is familiar with the ARM7TDMI core and ARM assembler. For more details on the ARM core architecture, refer to the ARM Developer Suite Guide and the ARM Technical Reference Manual. These documents are available from the ARM website.

1 EXCEPTION ENTRY AND EXIT

1.1 ENTERING AN EXCEPTION

This first section of this application note gives an introduction to Exception Handling by describing the basic default configuration of the ARM processors.

When an exception occurs, the ARM processor switches automatically to the ARM state and to the exception mode, copies the Current Program Status Register (CPSR) into SPSR_<mode> (where SPSR is the Saved Program Status Register, here in a particular mode), saves the return address in to LR_<mode>, sets the appropriate CPSR bits. The return address is stored in the Link Register, LR_<mode> to ensure that normal execution can resume following the exception handling. Finally, the Program Counter (PC) is set to the vector address to where the exception can be handled.

After handling the exception, the reverse process needs to be applied where CPSR is restored from SPSR_<mode> and the PC is restored from LR_<mode>.

Remember that exception handling can only be exited in ARM state, so if previously in Thumb state, prior to exit the exception, the core should switch states to ARM.

1.2 LEAVING AN EXCEPTION

To return from an exception, a data procession instruction is used however the exact instruction depends on the exception being handled.

In the ARM state, the use of the S bit normally sets the conditional flag. However, when in privileged modes, with the S bit set and the PC as the destination register, the instruction will update the PC and copy the SPSR_<mode> into the CPSR. Finally, still in privileged mode, LDM can be used with the ^ qualifier if LR_<mode> is adjusted before being stacked.

For SWI and Undefined exception handlers

```
MOVS  pc, lr
```

For FIQ, IRQ and Prefect Abort exception handlers

```
SUBS  pc, lr, #4
```

For Data Abort exception handlers

```
SUBS  pc, lr, #8
```

LDM can be used with the ^ qualifier if LR adjusted before being stacked

```
LDMFD  sp!,{pc}^
```

2 EXCEPTION PRIORITIES

It is important to remember that several exceptions can occur simultaneously. However, all exceptions are assigned priorities and are served in a predefined order.

The exception with the highest priority is Reset. Next, in order of decreasing priority, there is the Data Abort, FIQ (Fast), IRQ (Normal), Prefetch Abort, SWI, and finally the Undefined instruction with the lowest priority.

3 EXCEPTION VECTORS

The Reset Vector is at address 0x0000. This is followed by the undefined instruction vector, the SWI vector, the Prefetch Abort vector, Data abort vector, IRQ vector and FIQ vector.

The exception vector should contain a valid instruction to branch to the exception handler.

A branch instruction (B Exception_Handler) or data instruction (LDR pc,[pc,#offset]) may be used to jump to the exception handler.

The B instruction can be used if the exception handler is within 32MByte of the exception vector. The LDR pc,[pc,#offset] instruction loads a content of a memory to the PC, this memory points the exception handler. This memory location is pointed to by the contents of the program counter content plus an offset. This is done this way since the exception handler is outside the 32 Mbytes branch instruction range. Note also that the offset has to be within a 4Kbyte boundary, however, this is not a problem since this address itself doesn't need to contain any code.

The other method of accessing the exception handler is by using the MOV instruction. When the MOV PC instruction is used, only an appropriate address boundary can be used.

Specially for FIQ exception, since the FIQ vector is the last vector in the table, the handler can be written directly starting from the FIQ vector. This is quite normal for the FIQ handler, since it allows for it to be executed immediately, as fast as possible, without having to jump elsewhere.

4 ARM OR THUMB ?

It may be necessary to have an application which contains a mixture of ARM and Thumb code. When an exception occurs, the processor will switch the state automatically to ARM. This is because the instruction held in the vector table associated with the exception vector must be an ARM instruction. This vector instruction will jump the program counter to an exception handler. Once the execution of this exception handler has finished, the processor returns to execute the application in the state it was originally running in, either ARM or Thumb.

Note that the exception handler itself can be written in either ARM or Thumb instruction code. Any exception handler code written with the Thumb instruction set will return the processor to ARM state as the program counter will have to back to the original application code (i.e. main loop) with ARM instructions.

5 INTERRUPT HANDLING

This section of the application note gives an introduction to interrupt handling from the peripherals. In particular, the IRQ and FIQ interrupt modes are discussed. Many of the peripherals can be configured to generate interrupts in certain circumstances, for example a Timer might be set up to trigger an interrupt after a certain amount of time, periodically. How this Timer interrupt is handled by the exception handler is to be decided by the user-defined section of the handler of the code.

The ARM core has two levels of external interrupt handling, IRQ and FIQ, however the STR7 microcontrollers have a large number of interrupt sources. To manage all of the peripheral interrupts together with their priorities, the STR7 family of microcontrollers includes an Enhanced Interrupt Controller. The EIC has hardware handling of multiple interrupt channels, interrupt priority and automatic vectorization.

5.1 FIQ INTERRUPTS VERSUS IRQ INTERRUPTS

FIQ interrupts have a higher priority than IRQs which means that they will be serviced first when situations arise with both IRQ and FIQ events simultaneously. Servicing an FIQ will disable an IRQ and therefore IRQs will not be serviced until the FIQ handler completely exits.

FIQ interrupts are designed to be run as quickly as possible. The handler for FIQ can be placed directly at the end of the exception vectors table since the vector is the last in the table. A jump to another address therefore is not necessary and can be run immediately. In addition, FIQ mode has an additional 5 banked registers which means that there is less time wasted saving the non-banked registers before serving the FIQ exception handling and restoring them before exiting the exception.

5.2 SIMPLE INTERRUPT HANDLED IN C

The following example shows a simple interrupt handler written in C. In addition, the related assembly code is shown, compiled with and without __irq.

Note that the assembly code differs between one compiler to another.

```
__irq void IRQHandler (void)
{
u16 *IRQChannel = (u32 *)0xFFFFF804
 if (* IRQChannel == 0)                 // which interrupt was it
    Channel0_IRQHandler();              // process the interrupt
    // insert checks for other interrupt sources here
```

```
 *(IRQChannel+0x3C)=1<< (* IRQChannel ); // clear the interrupt
}
```

The next example shows the compiler output without the __irq keyword:

```
        STR       r14,[r13,#-4]!
        LDR       r2,0xe4c
        LDR       r0,[r2,#0]
        CMP       r0,#0
        BLEQ      Channel0_IRQHandler
        LDR       r0,[r2,#0]
        MOV       r1,#1
        MOV       r0,r1,LSL r0
        STR       r0,[r2,#0xf0]
        LDR       pc,[r13],#4
```

The compiler output with the __irq keyword:

```
        STRMFD    sp!,{r0-r4,r12,lr}
        LDR       r2,0xe4c
        LDR       r0,[r2,#0]
        CMP       r0,#0
        BLEQ      Channel0_IRQHandler
        LDR       r0,[r2,#0]
        MOV       r1,#1
        MOV       r0,r1,LSL r0
        STR       r0,[r2,#0xf0]
        LDMFD     sp!,{r0-r4,r12,lr}
        SUBS      pc,lr,#4
```

In the first example only the link register is saved to the stack then restored to the PC on return, however with the use of the __irq label, seven registers, including the link register, are saved to the stack. Finally and after restoring them, a data processing instruction, with the S bit set, is used to return from the exception handler.

6 INTERRUPT HANDLING USING THE EIC

6.1 EIC OVERVIEW

This section of the application note covers the IRQ handling using the EIC. The Enhanced Interrupt Controller has hardware handling of multiple interrupt channels, interrupt priority and automatic vectorization.

When a peripheral generates an interrupt, an IRQ line is set. If the corresponding bit is set in the Interrupt Enable Register (IER) then the related bit in the IPR register will be set to 1. A priority check by the EIC then follows to test if this interrupt priority is to be accepted. If the interrupt is accepted, the IRQ line is asserted low to the ARM core, and the IVR[15:0] register is updated with the contents of the SIRn[31:16] register for the related interrupt channel, resulting in a new Interrupt Vector.

When the IVR register is read, this informs the EIC that this interrupt is being processed, the IRQ line to the ARM core is asserted high. The details of the previous request need to be held so that processing can be resumed. Firstly, the EIC saves the value of the Current Interrupt Priority Register (CIPR) in the EIC Interrupt Priority in the Priority Stack, then updates CIPR with the new priority level. The priority stack can hold up to 16 entries.

Once the current interrupt being processed has finished, the related bit in the IPR register has to be cleared by software, the previous one can be restored for processing. The Priority level is restored back from the stack to CIPR, the channel ID is returned back and IVR creates the previous Interrupt Vector.

6.2 EIC METHODS

Several methods may be defined for the EIC to handle an IRQ interrupt, this application note will focus on three of them. These methods may be used for pointing a IRQ handler and depends on the IVR register content.

The first is the **address** method where a simple address is placed in IVR. For example, an IRQ handler can be placed on an appropriate address boundary pointed to directly by the direct memory address in IVR. The IVR[31:16] will contain the MSB of the interrupt handler address, while the SIR[31:16] will contain the LSB. In this case, an interrupt handler has to be located within 64KBytes starting from the base address defined in the IVR[31:16]. If IVR[31;16] = 0x6000, the first IRS may start at address 0x60000000, and the last one should start, at least, from address 0x6000FFFC.

Secondly, the **Branch** method puts a Branch instruction and IRQ handler where the handler is within the 32MBytes Branch instruction range. The IVR[31:16] will contain the MSB of the Branch instruction, while the SIR[31:16] will contain the Branch offset.

Lastly, the **LDR PC** method, a LDR pc,[pc,#offset] instruction can be used to set the program counter to load the value of an address nearby, where the address itself can be for a handler which is outside the 32Mbytes branch instruction range.

Note: when an EIC method is used, this will be fixed for all the interrupt channels.

6.3 NESTED INTERRUPT

Since the EIC manages the interrupt priority and automatic vectorization, nested interrupts are possible, however care must be taken to ensure that the system state is preserved when handling interrupts. For example, it is possible that the system state is lost if another interrupt preempts the current one when storing the link register (lr_irq) and the saved program status register (spsr_irq).

The issue lies in the fact that if care is not taken to correctly preserve the return address in the lr_irq register and the spsr_irq register contents, they may get corrupted if a second interrupt

occurs before the current interrupt has finished. The likely consequence is that the subroutine would return incorrectly.

The solution here is to disable IRQs in IRQ mode, save the lr_irq, spsr_irq and r0 to r12 registers to the IRQ stack, then change processor mode to system mode with IRQ enabled, so the whole interrupt service routine will be executed in system mode. However, it is crucial that, at the end of the handler the mode must be switched back to IRQ mode, the pending bit of the related current served channel is cleared in the IPR register and the Saved Program Status Register is restored in IRQ mode.

__irq therefore cannot be used to write a re-entrant IRQ handler, only a top level assembler should be used.

6.4 NESTED INTERRUPT EXAMPLE

The following example shows a top level assembler for nested interrupt handler when an instruction is used with IVR.

```
IRQHandler
        SUB    lr,lr,#4
        STMFD  sp!,{r0-r12,lr}
        MRS    r1,spsr
        STMFD  sp!,{r1}
        LDR    lr, =ReturnAddress
        LDR    pc, = IVR_addr
ReturnAddress
        LDR    r0, =EIC_Base_addr
        LDR    r2, [r0, #CICR_off_addr]
        MOV    r3,#1
        MOV    r3,r3,LSL r2
        STR    r3,[r0, #IPR_off_addr]
        LDMFD   sp!,{r1}
        MSR     spsr_cxsf,r1
        LDMFD   sp!,{r0-r12,pc}^

T0TIMIIRQHandler
        MSR    cpsr_c,#0x1F
        STMFD  sp!,{lr}
        BL     T0TIMI_IRQHandler
        LDMFD  sp!,{lr}
        MSR    cpsr_c,#0xD2
        MOV    pc,lr
```

Starting off with a program in System mode (in the case of an interrupt routine) or User mode (main program), suppose that an IRQ request is received and accepted by the EIC. Once the current instruction has finished executing, the ARM core switches to IRQ mode, saves the return address and cpsr to lr_irq and spcr_irq, then jumps to address 0x18 and the instruction in

the IRQ interrupt vector is executed and branches to the `IRQhandler` subroutine. During this time, IRQs are not allowed.

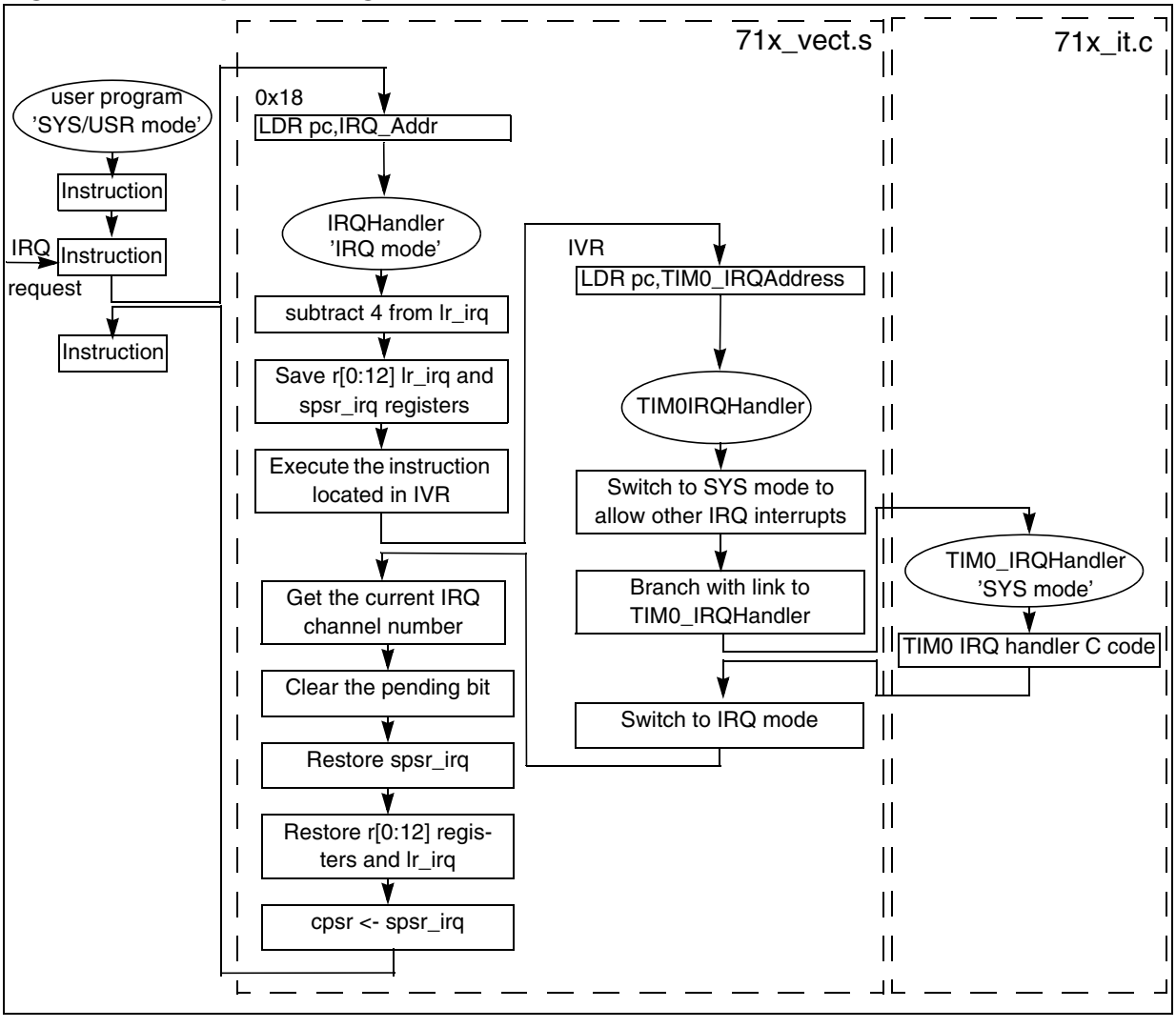
The IRQ handler starts by adjusting the lr_irq, then saves registers from 0 to 12, the spsr_irq and the lr_irq register to the IRQ stack, before executing the instruction located in the IVR, a return address is saved to the lr_irq. The IVR content is executed by pointing the PC to IVR register address.

The instruction in the IVR in this example calls the Timer0 Interrupt Handler (`T0TIMIIRQHandler`) where the mode is switched back into system mode to allow other IRQ interrupts once again. The full handler user code written in C, in this example is the `T0TIMI_IRQHandler` subroutine which can now be branched to. Now the core is in SYS mode so that the registers in IRQ mode are preserved.

Once this has finished, the processor returns and the core is switched back to IRQ mode, which once again disables other IRQs. Then the processor returns to the IRQHandler routine, the pending bit is cleared in the EIC_IPR register, the saved registers are restored, the program counter is restored from the link register and finally the cpsr is restored from the spsr of the IRQ mode.

Normal operation can then resume back in the user program with SYS/USR mode restored.

Figure 1. Interrupt Handling



6.5 EIC INITIALIZATION

The EIC initialization depends on the used EIC method. Source files are provided with this application note for the address method, Branch method and LDR PC method. The following top level assembler example shows how to configure the EIC for the LDR PC method.

```
;*******************************************************************************
;                        Exception vectors
;*******************************************************************************
        LDR     PC, Reset_Addr
        LDR     PC, Undefined_Addr
        LDR     PC, SWI_Addr
        LDR     PC, Prefetch_Addr
        LDR     PC, Abort_Addr
        NOP                             ; Reserved vector
        LDR     PC, IRQ_Addr
```

```
        LDR     PC, FIQ_Addr
;*******************************************************************************
;               Exception handlers address table
;*******************************************************************************
Reset_Addr      DCD     Reset_Handler
Undefined_Addr  DCD     UndefinedHandler
SWI_Addr        DCD     SWIHandler
Prefetch_Addr   DCD     PrefetchAbortHandler
Abort_Addr      DCD     DataAbortHandler
                DCD     0               ; Reserved vector
IRQ_Addr        DCD     IRQHandler
FIQ_Addr        DCD     FIQHandler

;*******************************************************************************
;              Peripherals IRQ handlers address table
;*******************************************************************************
;DCD Declares one or more words of store. In this example each DCD stores the
;address of a routine that handles a particular clause of the jump table.
;For more info refer to RealView Developer Kit, Assembler Guide.

T0TIMI_Addr     DCD  T0TIMIIRQHandler
FLASH_Addr      DCD  FLASHIRQHandler
RCCU_Addr       DCD  RCCUIRQHandler
RTC_Addr        DCD  RTCIRQHandler
WDG_Addr        DCD  WDGIRQHandler
XTI_Addr        DCD  XTIIRQHandler
USBHP_Addr      DCD  USBHPIRQHandler
I2C0ITERR_Addr  DCD  I2C0ITERRIRQHandler
I2C1ITERR_ADDR  DCD  I2C1ITERRIRQHandler
UART0_Addr      DCD  UART0IRQHandler
UART1_Addr      DCD  UART1IRQHandler
UART2_ADDR      DCD  UART2IRQHandler
UART3_ADDR      DCD  UART3IRQHandler
BSPI0_ADDR      DCD  BSPI0IRQHandler
BSPI1_Addr      DCD  BSPI1IRQHandler
I2C0_Addr       DCD  I2C0IRQHandler
I2C1_Addr       DCD  I2C1IRQHandler
CAN_Addr        DCD  CANIRQHandler
ADC12_Addr      DCD  ADC12IRQHandler
T1TIMI_Addr     DCD  T1TIMIIRQHandler
T2TIMI_Addr     DCD  T2TIMIIRQHandler
T3TIMI_Addr     DCD  T3TIMIIRQHandler
                DCD  0                  ; reserved
                DCD  0                  ; reserved
                DCD  0                  ; reserved
HDLC_Addr       DCD  HDLCIRQHandler
USBLP_Addr      DCD  USBLPIRQHandler
                DCD  0                  ; reserved
                DCD  0                  ; reserved
T0TOI_Addr      DCD  T0TOIIRQHandler
T0OC1_Addr      DCD  T0OC1IRQHandler
T0OC2_Addr      DCD  T0OC2IRQHandler

....
```

```
EIC_INIT
        LDR     r3, =EIC_Base_addr     ; Read the EIC base address
        LDR     r4, =0xE59F0000        ; Read the MSB of the LDR pc,[pc,#offset] opcode
        STR     r4, [r3, #IVR_off_addr]; Write the MSB of the LDR pc,[pc,#offset]
                                       ; instruction code in IVR[31:16]
                                       ; The LDR pc,[pc,#offset] opcode
                                       ; is : 0xE59FFxxx, with xxx the
                                       ; offset to the address pointing
                                       ; the interrupt handler.
                                       ; the LSB (Fxxx) will be computed and
                                       ; placed in to SIRn[31:16] registers
        LDR     r5, =SIR0_off_addr     ; Read SIR0 address
        LDR     r0, =T0TIMI_Addr       ; Read the start address of the IRQs
                                       ; address table. The IRQ handlers
                                       ; located ibelow the exception vectors,
                                       ; All addresses are 32bit long
        LDR     r1, =0x00000FFF
        AND     r0,r0,r1
        SUB     r4,r0,#8               ; subtract 8 for prefetch,
        LDR     r1, =0xF7E8            ; add the offset to the 0x00000000
                                       ; address(IVR address + 7E8 = 0x00000000)
                                       ; 0xFxxx used to complete the
                                       ; LDR pc,[pc,#offset] opcode
        LDR     r2, =32                ; 32 Channel to initialize
        ADD     r1,r4,r1               ; compute the jump offset
EIC_INI MOV     r4, r1, LSL #16        ; Left shift the result
        STR     r4, [r3, r5]           ; Store the result in SIRx register
        ADD     r1, r1, #4             ; Next IRQ address
        ADD     r5, r5, #4             ; Next SIR
        SUBS    r2, r2, #1             ; Decrement the number of SIR registers
                                       ; to initialize
        BNE     EIC_INI                ; If more then continue
...
```

Note: In order to run the examples attached to this application note, the STR71x RAM should be mapped to address 0x00000000.

7 REVISION HISTORY

Date	Revision	Changes
15-Jan-2004	1	Initial release.
28-Nov-2005	2	Title changed to "INTERRUPT HANDLING FOR STR7 MICROCONTROLLERS" Document content restructured

“THE PRESENT NOTE WHICH IS FOR GUIDANCE ONLY AIMS AT PROVIDING CUSTOMERS WITH INFORMATION REGARDING THEIR PRODUCTS IN ORDER FOR THEM TO SAVE TIME. AS A RESULT, STMICROELECTRONICS SHALL NOT BE HELD LIABLE FOR ANY DIRECT, INDIRECT OR CONSEQUENTIAL DAMAGES WITH RESPECT TO ANY CLAIMS ARISING FROM THE CONTENT OF SUCH A NOTE AND/OR THE USE MADE BY CUSTOMERS OF THE INFORMATION CONTAINED HEREIN IN CONNECTION WITH THEIR PRODUCTS.”

Information furnished is believed to be accurate and reliable. However, STMicroelectronics assumes no responsibility for the consequences of use of such information nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of STMicroelectronics. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. STMicroelectronics products are not authorized for use as critical components in life support devices or systems without express written approval of STMicroelectronics.

The ST logo is a registered trademark of STMicroelectronics.

All other names are the property of their respective owners

© 2005 STMicroelectronics - All rights reserved

STMicroelectronics group of companies

Australia – Belgium - Brazil - Canada - China – Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com